+++ date = '2025-02-21T10:19:46-08:00' draft = false title = 'Practica 3: Aplicación TODO en Haskell' +++

Paradigmas de la Programación

Práctica 3

Introducción

En este informe de práctica se detallará el proceso de creación de una aplicación de consola destinada a la gestión de listas de tareas (TODO list), desarrollada con el lenguaje de programación funcional **Haskell**. El propósito central fue adquirir conocimientos y familiarizarse con los principios básicos de Haskell, su sintaxis, el manejo de operaciones de entrada y salida, la manipulación de listas y el uso de herramientas como **Stack**. Esta actividad se llevó a cabo mediante la adaptación y el análisis de tutoriales y ejemplos existentes, con el fin de construir la aplicación de manera progresiva.

Descripción de la aplicación

La aplicación desarrollada permite a los usuarios gestionar una lista de tareas pendientes mediante una interfaz basada en la línea de comandos. Ofrece funciones esenciales como la creación, visualización, edición y eliminación de tareas.

Entre los comandos principales que admite la aplicación se encuentran:

- Añadir tarea (**+ texto_tarea**): Incorpora una nueva tarea a la lista.
- Listas tareas (**l**): Muestra todas las tareas pendientes, cada una identificada con un número de índice.
- Eliminar tarea (**- índice**): Suprime la tarea que corresponde al índice especificado.
- Editar tarea (**e índice**): Permite modificar el contenido de una tarea existente.
- Mostrar tarea (**s índice**): Muestra en pantalla una tarea específica.
- Limpiar lista (**c**): Elimina todas las tareas registradas en la lista.
- Salir (**q**): Finaliza la ejecución del programa.

Haskell y Stack: Entorno de desarrollo

Haskell es un lenguaje de programación puramente funcional. En el desarrollo de este proyecto se utilizó el compliador GHC (Glasgow Haskell Compiler), y se recurrió a **Stack** como herramientas principal para la gestión del proyecto, Stack permite crear proyectos de manera sencilla, gestionar sus dependencias , compilar y ejecutar el código, todo dentro de un entorno de desarrollo controlado. Además, garantiza la consistencia del entorno a encargarse de la instalación del compilador GHC y las bibliotecas necesarias de forma aislada para cada proyecto.

1. Creación del proyecto

El proyecto se inició en Haskell mediante el uso de Stack con el siguiente comando:

```
stack new MiProyectoTodo
cd MiProyectoTodo
```

2. Estructura del proyecto

La estructura generada por Stack organiza el código fuente, los archivos de prueba y la configuración del entorno de manera ordenada. Los elementos más relevantes para el desarrollo de esta práctica fueron:

- `app/Main.hs`: Archivo que contiene el punto de entrada principal de la aplicación.
- `src/` : Directorio destinado a los módulos de la biblioteca. En este poyecto, se creó el archivo **Crud.hs** para implementar la lógica de la aplicación.
- `test./Spec.hs`: Archivo reservado para la creación de prueba unitarias.
- `package.yaml`: Archivo de configuración de paquete. A partir de este, Stack genera el archivo `.cabal` automáticamente.
- `stack.yaml`: Archivo de configuración específico de Stack que define el entorno y las dependencias del proyecto.

Implementación de funcionalidades clave

La lógica principal de la aplicación se concentró en el módulo `Crud.hs`, mientras que el archivo `Main.hs` actuó como punto de entrada para iniciar la ejecución.

1. Módulo principal de lógica (`Crud.hs`)

Este módulo encapsula las funciones responsables de manejar el estado de la lista de tareas (representada como una lista de cadenas, `[string]`), así como la interacción con el usuario.

2. Bucle de interacción y procesamiento de comandos

La función principal `prompt`, definida en `Crud.hs`, controla el ciclo de vida de la aplicación. Su funcionamiento se basa en los siguientes pasos:

- Muestra un mensaje solicitando al usuario que ingrese un comando.
- Lee y procesa la entrada de usuario.
- Transfiere la interpretación del comando a una función llamada `interpret`.
- Llamada a sí misma recursivamente con la lista de tareas actualizada, repitiendo el proceso hasta que el usuario ingresa el comando `'q'` para salir.

```
-- Ejemplo simplificado del bucle en Crud.hs
module Crud (prompt) where

import System.IO (hFlush, stdout) -- Para flush de salida

type Tarea = String
type ListaTareas = [Tarea]

prompt :: ListaTareas -> IO ()
prompt tareas = do
    putStr "\nComandos (+, l, e, -, c, q): "
    hFlush stdout -- Asegura que el prompt se muestre antes de getLine
    comando <- getLine
    if comando == "q"
        then putStrLn "Adiós!"
```

```
        else procesarComando comando tareas

procesarComando :: String -> ListaTareas -> IO ()
procesarComando cmd tareas = do
    -- Aquí iría la lógica para interpretar 'cmd' y actualizar 'tareas'
    -- Por ejemplo:
    let nuevasTareas = case cmd of
                        ('+' : ' ' : desc) -> desc : tareas -- Añadir
                        "l" -> tareas -- Listar no cambia la lista aquí
                        _-> tareas -- Comando no reconocido
    if cmd == "l" then imprimirTareas nuevasTareas else return ()
    prompt nuevasTareas -- Llamada recursiva

imprimirTareas :: ListaTareas -> IO ()
imprimirTareas ts = mapM_ putStrLn $ zipWith (\n t -> show n ++ ". " ++ t) [0..]
ts
```

3. Gestión de tareas (Añadir, Listar, Eliminar, Editar)

- **Añadir**: Se implementó un patrón específico para detectar entradas con el formato `+ tarea` utilizando la coincidencia de patrón `'+': ' ':task`. Esto permite agregar la nueva tarea al inicio de la lista de tareas.
- **Listar**: Para mostrar todas las tareas con sus respectivos índices, se utilizó la función `zip` para emparejar cada tarea con un número consecutivo. Luego, se empleó `mapM_` para recorrer e imprimir cada par (índice, tarea) en la consola.
- **Eliminar**: Este comando requiere convertir la entrada del índice, que es una cadena (`string`), a un número entero (`int`). Para ello, se usó `readMaybe`, lo que añade seguridad al evitar errores por entradas no numéricas. Una vez validando el índice, se utilizó una función auxiliar que elimina el elemento correspondiente de la lista.

```
-- Ejemplo simplificado de función para eliminar
eliminarTarea :: Int -> ListaTareas -> Maybe ListaTareas
eliminarTarea idx tareas
    | idx < 0 || idx >= length tareas = Nothing -- Índice inválido
    | otherwise = Just (take idx tareas ++ drop (idx + 1) tareas)
```

- **Editar**: Similar a eliminar, se valida el índice y se remplaza la tarea correspondiente con una nueva entrada proporcionada por el usuario.

```
-- Ejemplo simplificado de función para editar
editarTarea :: Int -> String -> ListaTareas -> Maybe ListaTareas
editarTarea idx nuevoTexto tareas
    | idx < 0 || idx >= length tareas = Nothing
    | otherwise = Just (take idx tareas ++ [nuevoTexto] ++ drop (idx + 1) tareas)
```

- El manejo de `Maybe` es necesario para las operaciones que podrían fallar.

4. Punto de entrada de la aplicación (`Main.hs`)

El archivo `app/Main.hs` actúa como punto de entrada para la ejecución de la aplicación. En él, se importa la función `prompt` desde el módulo `Crud.hs` y se invoca con una lista de tareas vacía (`[]`). Además, muestra un mensaje de bienvenida al usuario antes de iniciar la interacción, dando comienzo al ciclo de comandos para la gestión de tareas.

```
-- Ejemplo simplificado de Main.hs
module Main where

import Crud (prompt) -- Asumiendo que Crud.hs está en src/

main :: IO ()
main = do
    putStrLn "--- Aplicación TODO en Haskell ---"
    putStrLn "Comandos disponibles: + tarea, l, e idx, - idx, s idx, c, q"
    prompt [] -- Iniciar con lista vacía
```

Pruebas Unitarias

El archivo `test/Spec.hs` se empleó para realizar pruebas a las funciones puras del proyecto, especialmente aquellas relacionadas con la manipulación de listas, como `eliminarTarea` y `editarTarea`. Estas pruebas se diseñaron para verificar que las funciones respondan correctamente tanto a entradas válidas como inválidas, asegurando así su correcto funcionamiento y la robustez del código ante posibles errores de uso.

```
-- Ejemplo simplificado de una prueba en Spec.hs
import Control.Exception (assert)
-- import Lib (editarTarea)

-- Función de prueba para editarTarea
testEdicion :: IO ()
testEdicion =
    let listaInicial = ["comprar pan", "estudiar Haskell"]
        resultadoEsperado = Just ["comprar leche", "estudiar Haskell"]
        resultadoReal = editarTarea 0 "comprar leche" listaInicial -- Usando una
función hipotética
    in assert (resultadoReal == resultadoEsperado) (putStrLn "Prueba de edición:
PASSED")

main :: IO ()
main = do
    putStrLn "Ejecutando conjunto de pruebas..."
    testEdicion
    putStrLn "Pruebas finalizadas."
```

Compliación y ejecución

La compliación del proyecto se lleva a cabo mediante el siguiente comando:

```
stack build
```

Para ejecutar la aplicación, se puede utilizar:

```
stack exec MiProyectoTodo-exe
```

O bien, ejecutar y compilar de forma combinada con:

```
stack run
```

Y para ejecutar las pruebas unitarias definidas en `text/Spec.hs`, se utiliza:

```
stack test
```

Conclusión

Esta práctica resultó para adquirir una comprensión básica pero sólida de Haskell y su ecosistema. A través del desarrollo de una aplicación TODO, aunque sencilla, se logró explorar conceptos clave del lenguaje funcional. Algunas complicaciones que tuve fueron las limitaciones técnicas como el almacenamiento disponible, lo que exigió soluciones creativas dento de los recursos disponibles.

Haskell proporciona una forma distinta de abordar problemas, y aunque su curva de aprendizaje inicial puede ser pronunciada, las ventajas que ofrece en términos de expresividad y robustez del código son notables. La experiencia obtenida, aunque introductoria, establece una base firme para continuar profundizando en el paradigma de la programación funcional en el futuro.

──୨ৎ──Repositorio ──୨ৎ──

⩾^- ᵕ -^⩽